Open access • Book Chapter • DOI:10.1007/978-3-642-33125-1_5

Modular static analysis with zonotopes — Source link

Eric Goubault, Sylvie Putot, Franck Védrine

Published on: 11 Sep 2012 - Static Analysis Symposium

Topics: Static analysis and Modular design

Modular Static Analysis with Zonotopes

Eric Goubault, Sylvie Putot, and Franck Védrine

CEA Saclay Nano-INNOV, CEA LIST, Laboratory for the Modelling and Analysis of
Interacting Systems, Point Courrier 174, 91191 Gif sur Yvette CEDEX,
Firstname.Lastname@cea.fr

Abstract. Being able to analyze programs function by function, or module by module is a key ingredient to scalable static analyses. The main difficulty for modular static analysis is to be able to do so while not losing too much precision. In this paper, we present a new summary-based approach that builds on previous work of the authors, a zonotopic functional abstraction, that is economical both in space and time complexity. This approach has been implemented, and experiments on numerical programs, reported here, show that this approach is very efficient, and that we still obtain precise analyses in realistic cases.

1 Introduction

In this paper, we use the particular properties that the zonotopic abstract domain [GP06,GGP09,GGP10] exhibits, to design a new modular static analysis of numeric properties. This domain has some advantages over the other sub-polyhedric abstract domains such as [Min01,SSM05,CH78], namely that its abstract transfer functions are of low complexity, while being more precise for instance for non-linear computations. This makes it a good candidate for scalable analyses of numeric properties. It has been the basis for the static analyzer FLUCTUAT, that extends this domain to deal with finite-precision arithmetic semantics (e.g. floating-point numbers) as in [GP11]. Experiments with FLUCTUAT [DGP$^+$09] have proved the usefulness of this domain for analysing mid-sized programs (up to 50KLoCs typically, on standard laptop computers). As we are dealing with precise numerical invariants (ranges of variables or functional properties, numerical errors and their provenance), the standard global interpretation of programs, re-analysing every function at each call site, may still prove too costly for analysing large programs of over 100KLoCs to several MLoCs.

But this zonotopic domain exhibits other properties that make it a perfect candidate for being used in modular static analyses: as shown in [GP09], our domain is a *functional* abstraction, meaning that the transfer functions we define abstract input/output relationships. This paper builds on this property, to design a precise and fast modular static analysis for numerical programs.

The program of Figure 1 will be used to exemplify the basic constructs in our modular analysis. Let us quickly sketch on this example, the behavior of the zonotopic modular abstraction that will be detailed in the rest of the paper. Intuitively, affine sets abstract a program variable x by a form $\hat{x} = \sum_i c_i^x \varepsilon_i + \sum_j p_j^x \eta_j$,

```
real mult(real a, real b)
  { return a*(b-2); }

compute(x ∈ [-1,1]);
```

```
real compute(real x)
  { real y1 = mult(x+1, x);
    real y2 = mult(x, 2*x);
    return y2-y1;
  }
```

Fig. 1. Running example

where c_i^x and p_j^x are real coefficients that define the abstract value, ε_i are symbolic variables with values in $[-1, 1]$ that abstract uncertain inputs and parameters, and η_j are symbolic variables with values in $[-1, 1]$ that abstract uncertainty on the value of x due to the analysis (i.e. to non-affine operations). The symbolic variables ε_i and η_j are shared by program variables, which implicitly expresses correlation. An affine form $\hat{x}$ is thus a function of the inputs of the program: it is a linear form of the noise symbols ε_i, which are directly related to these inputs.

Here, function `compute` is called with $\hat{x} = \varepsilon_1$ (input in [-1,1]). We build a summary for function `mult` after its first call (`y1 = mult(x+1, x);`). Using the semantics on affine sets to abstract the body of the function, we get as summary the ordered pair of affine sets (I, O) such that $I = (\varepsilon_1 + 1, \varepsilon_1), O = (-1.5 - \varepsilon_1 + 0.5\eta_1)$, where I abstracts the calling context and O the output.

At the next call (`y1 = mult(x, 2*x);`), we try to see if the previous summary can be used, that is if the calling context is contained in the input I of the summary: it is not the case as $(\varepsilon_1, 2\varepsilon_1) \leq (\varepsilon_1 + 1, \varepsilon_1)$ does not hold (with the order on affine sets defined by Equation 3).

We merge the two calling contexts with the join operator of Definition 4, and analyze again the body of the function: this gives a new (larger) summary for function `mult`: $I = (0.5+\varepsilon_1+0.5\eta_2, 1.5\varepsilon_1+0.5\eta_3), O = (-\frac{1}{4}-\frac{5}{4}\varepsilon_1-\eta_2+\frac{1}{4}\eta_3+\frac{9}{4}\eta_4)$.

Then, this new summary can be instantiated to the two calls (or any other call with calling context contained in affine set $I = (0.5 + \varepsilon_1 + 0.5\eta_2, 1.5\varepsilon_1 + 0.5\eta_3)$). Without instantiation, the output value of the summary ranges in $[-5, 4.5]$, using concretization of Definition 3. But the summary is a function defined over input ε_1 of the program, and over the symbols η_2 and η_3 that allow expressing the inputs of function `mult`: we will thus get a tighter range for the output, as well as a function of input ε_1, by instantiating η_2 and η_3 and substituting them in the output of the summary. For instance, for the second call of function `mult`, with $(\varepsilon_1, 2\varepsilon_1)$, we identify ε_1 with $0.5+\varepsilon_1+0.5\eta_2$ and $2\varepsilon_1$ with $1.5\varepsilon_1+0.5\eta_3$, and deduce $\eta_2 = -1$ and $\eta_3 = \varepsilon_1$, which yields for the output $\frac{3}{4} - \varepsilon_1 + \frac{9}{4}\eta_4 \in [-\frac{5}{2}, 4]$. Direct computation gives $1 - 2\varepsilon_1 + \eta_4 \in [-2, 4]$, which is just slightly tighter.

We illustrate this in Figure 2: we represent on the left picture, the calling contexts (a, b) for the two calls (I_1 is first call, I_2 is second call), and the zonotopic concretization of the calling context after merge, $I_1 \sqcup I_2$. On the right part of the figure, the parabola is the exact results of the second call, $ExM(I_2)$. The dashed zonotope is the result of the abstraction with the semantics of affine sets of the second call, $Mult(I_2)$. The zonotope in plain lines is the output of the summary, $Mult(I_1 \sqcup I_2)$. The zonotope in dotted lines is the summary instantiated to the second call.

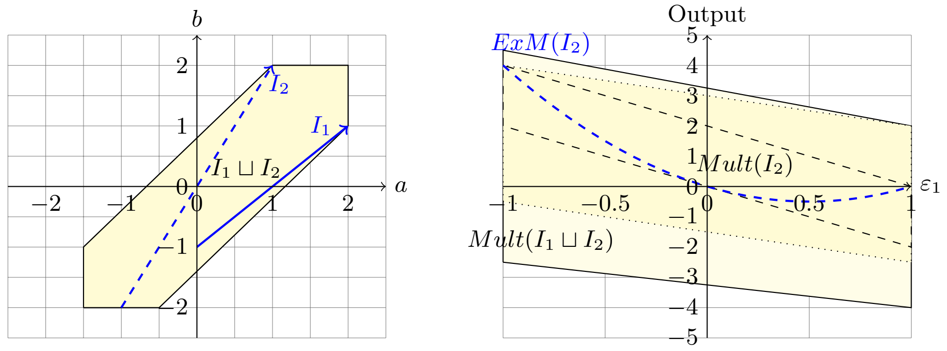


Fig. 2. Summary and instantiation (left is input, right output)

The performances of this modular analysis with our prototype implementation are demonstrated in Section 5.2.

Related work and contributions. Finding efficient ways to analyze inter procedural code is a long standing problem. A first class of methods consists in analyzing the whole control flow graph, re-analysing every procedure for each context. This may prove to be rather inefficient, as this may impose to analyze several times the same functions in very similar contexts. A second class considers separate analyses of procedures in a context-insensitive way. The same abstract return value is used for each call site. The advantage is that functions are only analyzed once, but the drawback is that the results that are used at each call site of a function may be a gross over-approximation of the correct result. This might lead to both imprecise and even time inefficient analyses, since imprecise abstractions might lead to lengthy imprecise least fixed point computations. Another approach is based on call strings abstractions [SP81]: the results of different calls to the same function are joined when the abstraction of the call stack (without considering environments, just the string of functions called) is the same. Among the classic abstractions of the call stack is the k-limiting abstraction, that considers equal all patterns of calls to functions that have their last k names of functions called, in order, equal. A way to improve on this is to use summary-based analyses. One creates an abstract value for each procedure, that summarizes its abstract transfer function, and which is instantiated for each call site. Most approaches use tabulation-based procedure summaries, see [SP81,CC77,RHS95,SRH96]. These tabulation-based approaches may be time and memory consuming while not always precise.

We develop here a context sensitive, summary-based approach, that corresponds to a symbolic relational separate analysis in the sense of [CC02], which is a relational function-abstraction in the sense of [JGR05]. The main originality of our work lies in the fact that we use a particular zonotopic domain [GP06,GP09], which abstracts functions (somewhat similarly, but in a much more efficient way, than the classic augmentation process with the polyhedric abstract domain [CC02]), and can thus be naturally instantiated. We will also briefly elaborate on improvements of our method, using a dynamic partitioning approach, as introduced in [Bou92]: one uses several summaries for a function, controlling their number by joining the closest (in a semantic sense) ones.

As already mentioned, the subject of modular analyses is huge, we mention here the closest work to ours. Some linear invariants are also found in a modular way in [MOS04]. Procedure summaries are inferred, but this time by using a backward analysis, in [GT07]. In the realm of pointer and shape analysis, which is orthogonal to our work, numerous techniques have been tried and implemented. See for instance [RC11,YYC08] for alias analysis, to mention but a few recent ones. Some complementary approaches can be found for instance in [Log07] for object-oriented features, and in [QR04] for dealing with concurrent programs.

Contents. We first state some of the basics of our zonotopic (or affine sets) functional abstract domain in Section 2. We describe in Section 3 how we create *summaries*, which associate to a given input context I, encoded as a zonotope, a zonotope O abstracting the input/output relationship, valid for all inputs that are included in I. We demonstrate how the input-output relationship abstracted in our zonotopic domain makes it very convenient to retrieve precise information on smaller contexts, through an instantiation process of the summary. We end up by presenting benchmarks in Section 5.

2 Functional Abstraction with Zonotopes

In this section, we quickly describe the abstract domain based on affine sets which is the basis for our modular analysis. Affine sets define an abstract domain for static analysis of numerical programs, based on affine arithmetic. The geometric concretization of an abstract value of affine sets is a zonotope, but the order we define on affine sets is stronger than the inclusion of the geometric concretization: it is equivalent to the inclusion of the zonotopes describing the abstract value and the inputs of the program. We thus get an input/output abstraction, which is naturally well suited for modular abstraction. The intersection, and thus the interpretation of tests, is a problematic operation: we partially by-pass this difficulty by enhancing our affine sets with constraints on the noise symbols [GGP10] used to define the affine sets. For a lighter presentation, the modular analysis will be presented here on affine sets without these constraints, but it can of course be used in the same manner with constrained affine sets.

2.1 Basics: Affine Sets and Zonotopes

Affine arithmetic is an extension of interval arithmetic on affine forms, first introduced in [CS93], that takes into account affine correlations between variables. An *affine form* is a formal sum over a set of *noise symbols* ε_i

$$\hat{x} \stackrel{\text{def}}{=} \alpha_0^x + \sum_{i=1}^{n} \alpha_i^x \varepsilon_i, \qquad (1)$$

with $\alpha_i^x \in \mathbb{R}$ for all i. Each noise symbol ε_i stands for an independent component of the total uncertainty on the quantity $\hat{x}$, its value is unknown but bounded in [-1,1]; the corresponding coefficient α_i^x is a known real value, which gives the magnitude of that component. The same noise symbol can be shared by several quantities, indicating correlations among them.

The semantics of affine operations is straightforward, they are exact in affine arithmetic. Non-affine operations are linearized, and new noise symbols are introduced to handle the approximation term. In our analysis, we indicate these new noise symbols as η_j noise symbols, thus introducing two kinds of symbols in affine forms of Equation 1: the ε_i noise symbols model uncertainty in data or parameters, while the η_j noise symbols model uncertainty coming from the analysis. For instance, the multiplication of two affine forms, defined, for simplicity of presentation, on ε_i only, writes

$$\hat{x}\hat{y} = \alpha_0^x \alpha_0^y + \sum_{i=1}^{n} \left(\alpha_i^x \alpha_0^y + \alpha_i^y \alpha_0^x\right) \varepsilon_i + \left(\sum_{i=1}^{n} |\alpha_i^x \alpha_i^y| + \sum_{i<j}^{n} |\alpha_i^x \alpha_j^y + \alpha_j^x \alpha_i^y| \right) \eta_1 .$$

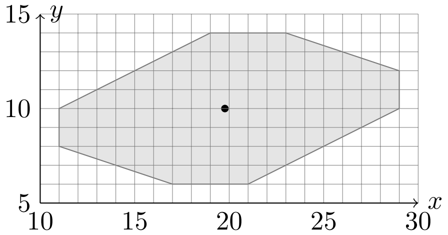
More generally, non-affine operations are abstracted by an approximate affine form obtained for instance by a first-order Taylor expansion, plus an approximation term attached to a new noise symbol. Affine operations have linear complexity in the number of noise symbols, whereas non-affine operations can be defined with quadratic cost.

Example 1. Let us demonstrate the abstraction on the following program:
`a = [-2,0]; b = [1,3]; x = a + b; y = -a; z = x * y;`
The assignments of a and b create new noise symbols ε_1, ε_2: $\hat{a} = -1 + \varepsilon_1$, $\hat{b} = 2+\varepsilon_2$. Affine expressions are handled exactly, we get $\hat{x} = 1+\varepsilon_1+\varepsilon_2$, $\hat{y} = 1-\varepsilon_1$. The multiplication produces a new η_1 symbol, we get $\hat{z} = 0.5 + \varepsilon_2 + 1.5\eta_1$. The range of z given by $\hat{z}$ is $[-2, 3]$ while the exact range is $[-2, 2.25]$.

In what follows, we introduce matrix notations to handle tuples of affine forms. We note $\mathcal{M}(n, p)$ the space of matrices with n lines and p columns of real coefficients. A tuple of affine forms expressing the set of values taken by p variables over n noise symbols $\varepsilon_i,\ 1 \le i \le n$, can be represented by a matrix $A \in \mathcal{M}(n + 1, p)$. Let ${}^tA$ denote the transpose of matrix A. We define the zonotopic concretization of such tuples by :

Definition 1. *Let a tuple of affine forms with p variables over n noise symbols, defined by a matrix $A \in \mathcal{M}(n + 1, p)$. Its concretization is the zonotope*

$$\gamma(A) = \left\{ {}^tA \begin{pmatrix} 1 \\ \varepsilon \end{pmatrix} \mid \varepsilon \in [-1, 1]^n \right\} \subseteq \mathbb{R}^p \ .$$



For instance, for $n = 4$ and $p = 2$, the gray zonotope is the concretization of the affine set $(\hat{x}, \hat{y})$, with $\hat{x} = 20 - 4\varepsilon_1 + 2\varepsilon_3 + 3\varepsilon_4$, $\hat{y} = 10 - 2\varepsilon_1 + \varepsilon_2 - \varepsilon_4$, and ${}^tA = \begin{pmatrix} 20 & -4 & 0 & 2 & 3 \\ 10 & -2 & 1 & 0 & -1 \end{pmatrix}$.

Now, we saw in the Definition of non-linear arithmetic operations, that our affine forms are defined over two kind of noise symbols, the ε_i and η_j. We thus define affine sets as Minkowski sums of a *central* zonotope, $\gamma(C^X)$ and of a *perturbation* zonotope centered on 0, $\gamma(P^X)$. Central zonotopes depend on central noise symbols ε_i, that represent the uncertainty on input values to the program, with which we want to keep as many relations as possible. Perturbation zonotopes depend on perturbation symbols η_j which are created along the interpretation of the program and represent the uncertainty of values due to operations that are not interpreted exactly: for instance the control-flow abstraction while computing the join of two abstract values, or non-affine arithmetic operations.

Definition 2. *We define an affine set by the pair of matrices* $X = (C^X, P^X) \in \mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$.
The affine form $X_k = c^X_{0k} + \sum_{i=1}^n c^X_{ik}\varepsilon_i + \sum_{j=1}^m p^X_{jk}\eta_j$ *describes its kth variable.*

2.2 Geometric and Functional Orders

Definition 3. *Let* $X = (C^X, P^X)$ *be an affine set in* $\mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$. *Its concretization in* $\mathcal{P}(\mathbb{R}^p)$ *is*

$$\gamma(X) = \left\{ {}^tC^X \begin{pmatrix} 1 \\ \varepsilon \end{pmatrix} + {}^tP^X \eta \mid (\varepsilon, \eta) \in [-1, 1]^{n+m} \right\} .$$

If we were only interested in abstractions of current values of variables, the partial order to consider would be the subset inclusion of their concretization, as formalized in Definition 3. But we are interested in abstracting input/output relations, this will be instrumental in our modular analysis.

Let $\mathcal{X}$ be a set of functions of the form $x : \mathbb{R}^q \to \mathbb{R}^p$. We write $x_1, \ldots, x_p$ its p components. Our goal is to abstract the input/output relationship of functions in $\mathcal{X}$ using an affine set X, i.e. to automatically determine an over-approximation X of the set of values that $e_1, \ldots, e_q, x_1, \ldots, x_p$ can take, conjointly, where $e_1, \ldots, e_q$ are slack variables representing the initial values of the q input variables of functions $x \in \mathcal{X}$: to one particular run of the program, corresponds exactly one fixed tuple of values $e_1, \ldots, e_q$. This fits in the *relational function-abstraction* of [JGR05]. Let $\gamma(\tilde{X})$ be such an augmented zonotope, $\tilde{X} \in \mathcal{M}(r, p+q)$, where the set of symbols is decomposed in $r = n + m$ symbols $\varepsilon_1, \ldots, \varepsilon_n$, the central noise symbols, and $\eta_1, \ldots, \eta_m$, the perturbation symbols, as introduced in Definition 2. From now on, we will consider the augmented affine set $\tilde{X}$:

$$\tilde{X} = \begin{pmatrix} E & C^X \\ 0 & P^X \end{pmatrix} \tag{2}$$

where $E \in \mathcal{M}(n+1,q)$ is the affine set describing the inputs to the functions in $\mathcal{X}$. The concretization γ_f of such augmented affine sets in terms of sets of functions from $\mathbb{R}^q$ to $\mathbb{R}^p$ is as follows:

$$\gamma_f(\tilde{X}) = \left\{ f : \mathbb{R}^q \to \mathbb{R}^p \mid \begin{array}{c} \forall \varepsilon \in [-1,1]^n, \exists \eta \in [-1,1]^m, \\ f({}^tE \begin{pmatrix} 1 \\ \varepsilon \end{pmatrix}) = {}^tC^X \begin{pmatrix} 1 \\ \varepsilon \end{pmatrix} + {}^tP^X \eta \end{array} \right\}$$

The (partial) order relation on augmented affine sets $\tilde{X}$, $\tilde{Y}$, is given by: $\tilde{X} \leq_f \tilde{Y}$ if $\gamma_f(\tilde{X}) \subseteq \gamma_f(\tilde{Y})$, which in turn is equivalent to $\gamma(\tilde{X}) \subseteq \gamma(\tilde{Y})$, hence correctness of our functional abstraction is given naturally as for any concretization-based abstract interpretation [CC92]: $\tilde{X}$ is a correct abstraction of a set $\mathcal{X}$ of functions if $\mathcal{X} \subseteq \gamma_f(\tilde{X})$. Then, similarly for the interpretation of an abstraction F of a function $\mathcal{F}$ on augmented affine sets: $\forall \mathcal{X} \in \mathbb{R}^{n+p}$, $\mathcal{F}(\mathcal{X}) \subseteq \gamma(F(\tilde{X}))$.

Now, the order relation on augmented affine sets can be reformulated in terms of the current parameterization of abstract values for variables $x_1, \ldots, x_p$, without having to consider the extra n variables $e_1, \ldots, e_n$: let X and Y be two affine sets. We say that $X \leq Y$ iff for all $t \in \mathbb{R}^p$,

$$\|(C^Y - C^X)t\|_1 \leq \|P^Y t\|_1 - \|P^X t\|_1 \ . \tag{3}$$

This functional (pre-)order $\leq$ always implies $\leq_f$, and is equivalent in most interesting situations, for instance when matrix E of equation 2, without its first line, is invertible: this covers in particular the case when the inputs are given in intervals and have unknown dependency. We do not prove this property here as this is not central to the rest of the paper, some hints about it can be found in [GP09].

Example 2. Take $X : (X_1 = \varepsilon_1, X_2 = \varepsilon_2)$ and $Y : (Y_1 = \varepsilon_2, Y_2 = \varepsilon_1)$. We have $\gamma(X) = \gamma(Y) = [-1,1]^2$. But X and Y are incomparable for the functional ordering of Equation 3. Indeed, X and Y represent two very different functions from the inputs $(\varepsilon_1, \varepsilon_2)$ to the values of the variables (x_1, x_2).

2.3 Join Operation

In general, there exists no least upper bound for affine sets. We define a join operator over affine sets which gives a minimal upper bound in some cases, can always be computed efficiently, and presents some nice properties: for instance, the range of the joined value on each variable is equal to the union of the interval ranges on the variable. We refer the reader to [GP09] for details.

Let us first introduce some notations. For two real numbers α and β, let $\alpha \wedge \beta$ denote their minimum and $\alpha \vee \beta$ their maximum. We define

$$\text{argmin}_{|.|}(\alpha, \beta) = \gamma \text{ such that } \gamma \in [\alpha \wedge \beta, \alpha \vee \beta] \text{ and } |\gamma| \text{ is minimal}$$

Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two intervals. We say that $\boldsymbol{x}$ and $\boldsymbol{y}$ are in generic positions if, whenever $\boldsymbol{x} \subseteq \boldsymbol{y}$, $\inf \boldsymbol{x} = \inf \boldsymbol{y}$ or $\sup \boldsymbol{x} = \sup \boldsymbol{y}$. And for an interval $\boldsymbol{x}$, we note $mid(\boldsymbol{x})$ its center.

Definition 4. *Let two affine sets X and Y where (C^X, P^X) and (C^Y, P^Y) are in $\mathcal{M}(n+1,p) \times \mathcal{M}(m,p)$, we define $Z = X \sqcup Y$ such that for all $k, l \in [1,p]$, $i \in [1,n]$, $j \in [1,m]$:*
If $\gamma(X_k)$ and $\gamma(Y_k)$ are in generic position:

$$\begin{aligned} c^Z_{0,k} &= mid\,(\gamma(X_k) \cup \gamma(Y_k)) \\ c^Z_{i,k} &= argmin_{|.|}(c^X_{i,k}, c^Y_{i,k}), p^Z_{j,k} = argmin_{|.|}(p^X_{j,k}, p^Y_{j,k}) \\ p^Z_{m+k,k} &= \sup(\gamma(X_k) \cup \gamma(Y_k)) - c^Z_{0,k} - \left(\sum_{i=1}^{n} \mid c^Z_{i,k} \mid + \sum_{j=1}^{m} \mid p^Z_{j,k} \mid \right) \end{aligned}$$

$$\textit{Else: } c^Z_{0,k} = \frac{c^X_{0,k} + c^Y_{0,k}}{2}, c^Z_{i,k} = \frac{c^X_{i,k} + c^Y_{i,k}}{2}, p^Z_{j,k} = \frac{p^X_{j,k} + p^Y_{j,k}}{2}$$

$$\boldsymbol{p}^Z_{m+k,k} = \frac{1}{2} \sum_{i=0}^{n} |c^Y_{i,k} - c^X_{i,k}| + \frac{1}{2} \sum_{j=1}^{m} |p^Y_{j,k} - p^X_{j,k}|$$

And in both cases: $p^Z_{m+l,k} = 0$ for $l \neq k$

Intuitively, by using the *argmin* operator, this join operator keeps the dependencies to the inputs that are common to both form joined.

We then have the following result (whose second item is proved in [GP09]):

Lemma 1. *$Z = X \sqcup Y$ is an upper bound of X and Y such that:*

- *for all $k \in [1,p]$, Z_k is a minimal upper bound of X_k and Y_k*
- *if X_k and Y_k are in generic positions, then $k \in [1,p]$, $\gamma(Z_k) = \gamma(X_k) \cup \gamma(Y_k)$ where $\cup$ is here the union in the lattice of intervals.*

Example 3. Take $X : (X_1 = 1 + \varepsilon_1, X_2 = \varepsilon_1)$ and $Y : (Y_1 = 2\varepsilon_1, Y_2 = \varepsilon_1)$. We have $\gamma(X_1) = [0,2]$, $\gamma(Y_1) = [-2,2]$, so that X_1 and Y_1 are in generic positions. Then $Z = X \cup Y : (Z_1 = \varepsilon_1 + \eta_1, Z_2 = \varepsilon_1)$ is a minimal upper bound of X, Y.

3 Affine Sets Summary and Specialization

We will now define function summaries as pairs (I, O) of input and output zonotopes, I and O being defined as introduced in Section 2. These zonotopes I and O are parametrized by the same central noise symbols $\varepsilon_1, \ldots, \varepsilon_n$ representing the inputs of the program, and thus each represent a function of these inputs. But the pair also represents functions from $\gamma(I)$ to $\gamma(O)$, and we will introduce a new functional concretization $\gamma_f(I, O)$ that extends the γ_f of Section 2.

Pairs (I, O) abstract sets of functions from $\gamma(I)$ to $\gamma(O)$, deduced from I and O seen as sets of functions of the inputs of the program (the noise symbols ε_i), and of uncertainties introduced by the analysis (the noise symbols η_j). Indeed, as O represents some computation on entries I, O contains the perturbation symbols of I: say $\eta_1, \ldots, \eta_{m_1}$ for I, $\eta_{m_1+1}, \ldots, \eta_m$ for the symbols only appearing in O. More formally, the concretization of a pair (I, O) of input and output zonotopes, in terms of functions F from $\gamma(I)$ to $\gamma(O)$, is as follows:

$$\gamma_f(I,O) = \left\{ \begin{array}{r} F : \gamma(I) \rightarrow \gamma(O) \mid \forall \varepsilon_1, \ldots, \varepsilon_n, \eta_1, \ldots, \eta_{m_1} \\ \exists \eta_{m_1+1}, \ldots, \eta_m \text{ with } (\varepsilon, \eta) \in [-1,1]^{n+m} \\ \text{and } F({}^tI^t(1, \varepsilon, \eta_1, \ldots, \eta_{m_1})) = {}^tO^t(1, \varepsilon, \eta) \end{array} \right\}$$

As outputs are defined over these same noise symbols, the summaries can be instantiated to a given calling context, by substituting some of these perturbation noise symbols by their expression for the particular calling context.

Consider a current calling context C, and a current function summary $S_f = (I, O)$ for f, the interpretation of the function call $f(C)$ in our inter-procedural analysis is given in Algorithm 1. Its different steps are detailed in the sections that follow.

Algorithm 1 Interpretation of function call $f(C)$, given calling context C, and function summary $S_f = (I, O)$

```
if  $!(C \leq I)$ // test if calling context $C$ matches summary input $I$ then
    $I \leftarrow I \sqcup C$ // join calling context and summary input (Definition 4)
    $S_f \leftarrow (I, [\![f]\!](I))$ // new summary creation (Section 3.2)
end if
return $[\![I == C]\!]O$ // summary instantiation (Section 3.3)
```

3.1 Program Syntax and Semantics

Programs *Prog* we are considering in what follows are sets of functions $f \in Prog$, acting on an environment made up of variables $\mathcal{V}_f$ local to function f, and global variables $\mathcal{G}$. We suppose that the $\mathcal{V}_f$, $f \in Prog$, and $\mathcal{G}$, form a partition of the set of program variables $\mathcal{V}$. There is a unique data type: the real numbers. Function definitions are as follows:

$$\begin{array}{l} funct = \text{function } f(v_1, \ldots, v_p) \ \{ \\ \qquad\qquad instr; \ \text{return } r \ \} \qquad v_1, \ldots, v_p \in \mathcal{V}_f, \ r \in \mathcal{V}_f \end{array}$$

Function calls $f(expr_1, \ldots, expr_p)$, where $expr_1$ to $expr_p$ are p expressions, have the call by value semantics: their evaluation correspond to computing the value of each expression $expr_1$ to $expr_p$ in that order, and assigning each local variable v_i with the corresponding value of $expr_i$ ($i = 1, \ldots, p$) in the environment of the call. The body of the functions is standard, it is made of a classic set of instructions *instr* for imperative languages: assignment of expressions to variables, tests, loops. The *return* at the end of the definition of f just returns the value of one of the local variables, r, of f, to the caller. We consider global variables as part of the calling context and output of functions. A function f is thus defined from $\mathbb{R}^p$ to $\mathbb{R}^q$, for some $q \leq card(\mathcal{G}) + 1$.

We suppose given a set of control points attached to instructions of our language, including $(call^i_f)$ just before executing the ith call to f: $f(expr, \dots, expr)$ in an expression, and $(return^i_f)$, right after the ith call to function f has returned the flow of execution to the calling expression.

The concrete collecting semantics is given in terms of concrete environments $e \in Env = \mathcal{V} \to \mathbb{R}$, and a semantics function, partitioned over the control points $d \in \mathcal{D}$, $[\![instr]\!]_c : Env \to Env$, such that $[\![E]\!]^d_c e$ ($E \in instr$, $e \in Env$) gives the change of concrete environment when interpreting instruction E in context e, at control point d. The concrete collecting semantics $[\![P]\!]_c e$, partitioned over $d \in \mathcal{D}$ as $[\![P]\!]^d_c e$, of a program P, is obtained as the least solution in $\wp(Env)$ (with subset ordering), over some set of initial environments $e \in \wp(Env)$, of the semantic equations given by $[\![E]\!]^d_c$ lifted from Env to $\wp(Env)$, at each control point of P.

For the abstract collecting semantics, we use the abstract domain of affine sets $\mathcal{Z}$ for all instructions, except for calls to functions. In order to define a modular static analysis, we suppose now abstract environments in Env_a are made of bindings of variables to affine sets, as well as bindings of function names to summaries, i.e. pairs of affine sets: $Env_a = \mathcal{Z} \times (Prog \to \mathcal{Z} \times \mathcal{Z})$. We call $[\![instr]\!]^d_a : Env_a \to Env_a$ the corresponding semantics functions (forward abstract transformers). The correctness of the abstract semantics with respect to the concrete one is formalized as follows. For all initial possible sets of environments $e \in \wp(Env)$, we form $e^\sharp = (e^\sharp_{\mathcal{V}}, \bot)$ where $e^\sharp_{\mathcal{V}}$ is any abstract environment in $\mathcal{Z}$, dealing only with program variables, such that $e \in \gamma(e^\sharp_{\mathcal{V}})$, and $\bot$ in the second component of $e^\sharp$ means that for all functions f of $Prog$, we start in an environment where we do not have any summary of f, then we must have, for all control points $d \in \mathcal{D}$:

$$[\![P]\!]^d_c e \subseteq \gamma\left(\pi_1([\![P]\!]^d_a e^\sharp)\right) \tag{4}$$

where $\pi_1 : Env_a \to \mathcal{Z}$ is the projection on the first component of environments.

3.2 Summary Creation

The operations involved, order $\leq$ and join, have already been described in Section 2. Let us just here consider Example 1. Function `compute` is called with $x \in [-1, 1]$, which can be abstracted by $x = \varepsilon_1$. The first call to function `mult` is then interpreted in Algorithm 1 as a new summary creation, since the current summary for `mult` is $\bot$. We thus interpret `mult` with arguments $a_1 = 1 + \varepsilon_1$ and $b_1 = \varepsilon_1$. Multiplication $a_1 \times (b_1 - 2)$ in the abstract domain of affine sets produces $a_1 \times (b_1 - 2) = -1.5 - \varepsilon_1 + 0.5\eta_1$, where η_1 is a new noise symbol with values in $[-1, 1]$. The abstract environment at the end of this first call to `mult` contains the entry, for `mult` $(I = (a_1 = 1 + \varepsilon_1, b_1 = \varepsilon_1), O = -1.5 - \varepsilon_1 + 0.5\eta_1)$.

3.3 Summary Instantiation

The instantiation of a summary for a given calling context resembles the meet operation on constrained affine sets [GGP10]: indeed, it consists in adding constraints on noise symbols that correspond to component-wise equality of affine forms. Still, it does not require the formalism of constrained affine sets as we do not abstract constraints, they are immediately used to substitute in the summary the noise symbols introduced by the join operation due to merging contexts, by the affine expression of the other noise symbols.

The instantiation operator is thus a function that takes a summary (I, O), an input affine set C such that $C \leq I$ and returns $Z = [\![I == C]\!]O$. We form the following matrix U, given $I = (C^I, P^I)$ and $C = (C^C, P^C)$ two affine sets with $(C^I, P^I), (C^C, P^C) \in \mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$:

$$U = \begin{pmatrix} p^I_{m,1} - p^C_{m,1} \dots p^I_{1,1} - p^C_{1,1} \; c^I_{n,1} - c^C_{n,1} \dots c^C_{0,1} - c^C_{0,1} \\ \dots \\ p^I_{m,p} - p^C_{m,p} \dots p^I_{1,p} - p^C_{1,p} \; c^I_{n,p} - c^C_{n,p} \dots c^C_{0,p} - c^C_{0,p} \end{pmatrix}$$

Performing Gauss elimination [Bee06], on U we obtain the row-echelon form for U: $U' = {}^t(U_1|U_2)$ where $U_1 \in \mathcal{M}(n+m+1, r)$ and $U_2 \in \mathcal{M}(n+m+1, p-r)$ with $r = min(m, p)$ and U_1 and U_2 upper triangular. Matrix U_1 encodes the fact that we must have, when "interpreting" $I == C$, relations of the form $\eta_{k_1} = R_1(\eta_{k_1-1}, \dots, \eta_1, \varepsilon_n, \dots, \varepsilon_1)$, $\eta_{k_2} = R_2(\eta_{k_2-1}, \dots, \eta_1, \varepsilon_n, \dots, \varepsilon_1)$, ..., $\eta_{k_r} = R_r(\eta_{k_r-1}, \dots, \eta_1, \varepsilon_n, \dots, \varepsilon_1)$, with $k_1 > k_2 > \dots k_r$.

The principle of the instantiation operator defined below is, first, to interpret the relation $U^t(\eta_m, \dots, \eta_1, \varepsilon_n, \dots, \varepsilon_1) = 0$ as constraints on the values taken by noise symbols, and to use the r relations $R_1, \dots, R_{k_r}$ to eliminate the perturbation symbols that have been introduced the most recently in the summary output O of function f, $\eta_{k_1}, \dots, \eta_{k_r}$:

Definition 5. *Let $I = (C^I, P^I)$, $C = (C^C, P^C)$ and $O = (C^O, P^O)$ be three affine sets with $(C^I, P^I), (C^C, P^C) \in \mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$ and $(C^O, P^O) \in \mathcal{M}(n+1, q) \times \mathcal{M}(m, q)$ (by convention, m is the total number of perturbation noise symbols, some lines in P^I, P^C and P^O may contain only zeros). In $Z = [\![I == C]\!]O$, the (C^Z, P^Z) are defined by substituting in (C^O, P^O) the values of η_{k_1} to η_{k_r} given by $R_1(\eta_{k_1-1}, \dots, \eta_1, \varepsilon_n, \dots, \varepsilon_1)$ to $R_{k_r}(\eta_{k_r-1}, \dots, \eta_1, \varepsilon_n, \dots, \varepsilon_1)$ respectively, in terms of the η_j of lower indices, and of the ε_j.*

In practice, there is actually no need to first perform the Gauss elimination on U: constraints $I == C$ are of such a particular form that it is enough (and this is what is implemented and tested in this article) to substitute in (C^O, P^O), in order to obtain (C^Z, P^Z), only the relations in U that are already in row-echelon form.

Note that when a function contains only operations that are affine with respect to the calling context (no join operation), the instantiation of the summary to a particular context gives the same result as would do the direct abstraction of the function in that context. When non-affine operations are involved, such as in the running example, we will see that the instantiation also gives tight results.

Let us consider the second call to function `mult` in our running example, with $a_2 = \varepsilon_1$ and $b_2 = 2\varepsilon_1$. This calling context is not included in the previous one, so in Algorithm 1 we need first to merge the current summary input context I

with the current call context, giving $(a_3, b_3) = (a_2, b_2) \sqcup (a_1, b_1) = (0.5 + \varepsilon_1 + 0.5\eta_2, 1.5\varepsilon_1 + 0.5\eta_3)$ (note that we are in the non-generic case of Definition 4). The zonotopic concretizations of the two calling contexts, and of the merged value giving the input context of the summary, are represented on the left picture of Figure 2 (and respectively named I_1, I_2 and $I_1 \sqcup I_2$).

The result of the multiplication for the merged context is then $a_3 \times (b_3 - 2) = -\frac{1}{4} - \frac{5}{4}\varepsilon_1 - \eta_2 + \frac{1}{4}\eta_3 + \frac{9}{4}\eta_4 \in [-5, \frac{9}{2}]$, so we create the new summary for `mult`: $(I = (a_3 = 0.5 + \varepsilon_1 + 0.5\eta_2, b_3 = 1.5\varepsilon_1 + 0.5\eta_3), O = -\frac{1}{4} - \frac{5}{4}\varepsilon_1 - \eta_2 + \frac{1}{4}\eta_3 + \frac{9}{4}\eta_4)$. This is the zonotope (parallelepiped, here), represented in plain lines on the right picture of Figure 2.

Let us now instantiate this summary for the second call (last part of Algorithm 1), when $a = \varepsilon_1 = 0.5 + \varepsilon_1 + 0.5\eta_2$ and $b = 2\varepsilon_1 = 1.5\varepsilon_1 + 0.5\eta_3$: we deduce by elimination $\eta_2 = -1$ and $\eta_3 = \varepsilon_1$. Instantiating the summary with these values of η_2 and η_3 yields $a_3 \times (b_3 - 2) = \frac{3}{4} - \varepsilon_1 + \frac{9}{4}\eta_4 \in [-\frac{5}{2}, 4]$. While direct computation $a_2 \times (b_2 - 2) = 1 - 2\varepsilon_1 + \eta_4 \in [-2, 4]$, which is just slightly tighter. The instantiated and directly computed zonotopes are also represented on the right picture of Figure 2, respectively in dotted and dashed lines.

3.4 Correctness and Complexity

The correctness of instantiation comes from the fact that, writing $F_{|\gamma(C)}$ for the restriction of a function $F : \gamma(I) \to \gamma(O) \in \gamma_f(I, O)$ (remember that $C \leq I$, hence $\gamma(C) \subseteq \gamma(I)$), first, $F(\gamma(C)) \subseteq \gamma(Z)$ and

$$\{F_{|\gamma(C)} : \gamma(C) \to \gamma(Z) | F \in \gamma_f(I, O)\} \subseteq \gamma_f(C, Z) \tag{5}$$

The last statement meaning that (C, Z) is a correct summary for restrictions of functions F summarized by the more general summary (I, O).

We can conclude from this that Algorithm 1 is correct in the sense of Equation 4. This is done by showing inductively on the semantics that, for any program P, and for any concrete environment e and abstract environment $e^\sharp$ such as in the premises of Equation 4,

$$\{F | \forall x \in [\![P]\!]_c^{(call_g^i)} e, F(x) = [\![g]\!]_c^{(return_g^i)} x\} \subseteq \gamma_f \left(\pi_2 \left([\![P]\!]_a^{(return_g^i)} e^\sharp \right) (g) \right) \tag{6}$$

$$[\![P]\!]_c^{(call_g^i)} e \subseteq \gamma \left(\pi_1 \left([\![P]\!]_a^{(call_g^i)} e^\sharp \right) \right) \tag{7}$$

In Equation 6, $\pi_2 : (Env_a = Z \times (Prog \to Z \times Z)) \to (Prog \to Z \times Z)$, extracts the part of the abstract semantics which accounts for the representation of summaries. The fact that Equation 6 is true comes from the fact that summary instantiations compute correct over-approximations of concrete functions on the calling contexts, see Equation 5. The fact that Equation 7 is true comes from the fact that in Algorithm 1 we join the calling context with the current summary context as soon as it is not included in it, so we safely over-approximate all calling contexts in the abstract fixed-point.

Without the inclusion test, the complexity of Algorithm 1 and of (the simpler version of) summary instantiation is $O(q \times nb\_noise)$, where q is the number of arguments of the called function and nb_noise is the number of noise symbols involved in the affine form of the calling context. The main contribution to the complexity comes from the (particularized) Gauss elimination in the substitution. The inclusion test is by default exponential in the number of noise symbols. However, simpler $O(q \times nb\_noise)$ versions can be used as a necessary condition for inclusion, and in many cases the full test can be avoided, see [GP09]. Also, the use of a modular abstraction as proposed here helps to keep the number of noise symbols quite low.

4 Summary Creation Strategies

In order to control the loss of accuracy due to summarizing, it is natural to use several (a bounded number n of) summaries instead of a unique one. We present here a method inspired from tabulation-based procedure summaries [SP81] and dynamic partitioning [Bou92].

We consider a function call $f(C)$, when k summaries (I_j, O_j), $j = 1, \ldots, k$ exist for function f. Either C is included in one of the inputs I_j, and the call reduces to instantiating O_j and thus returning $[\![I_j == C]\!]O_j$. Or else, we distinguish two cases. If the maximal number n of pairs (I_j, O_j) is not reached, we add a new summary $(C, [\![f]\!](C))$. And if it is reached, we take the closest calling context I_j to C (such that the cost $c(C, I_j)$ is minimal) and replace in the table of summaries, (I_j, O_j) by the new summary $(I_j \sqcup C, [\![f]\!](I_j \sqcup C))$.

For instance, the cost function c could be chosen as follows: let $(e_l)_{1 \leq l \leq p}$ be the canonical basis of $\mathbb{R}^p$,

$$c(X,Y) = \sum_{l=1}^{p} \left(\|(C^Y - C^X)e_l\|_1 - \mid \|P^Y e_l\|_1 - \|P^X e_l\|_1 \mid \right) \ .$$

By definition of the order relation 3, if $X \leq Y$ or $Y \leq X$ then $c(X,Y) \leq 0$. This function, which defines a heuristic to create or not new summaries, expresses a cost function on the component-wise concretizations of the affine sets X and Y. And it can be computed more efficiently (in $O(p(n+m))$ operations, where p is the number of variables and $n+m$ the number of noise symbols) than if we used a stronger condition linked to the order on the concretizations of these affine sets.

5 Examples

We have implemented the zonotopic summarizing in a small analyzer of C programs. The part dedicated to zonotopes and summarizing represents about 7000 lines of C++ code. We present experiments showing the good behavior in time and accuracy of summaries, then some results on realistic examples. The examples are available on `http://www.lix.polytechnique.fr/Labo/Sylvie.Putot/benchs.html`.

5.1 Performance of the Modular Analysis

To evaluate the performance of the summarizing process, we slightly transform the running example of Figure 1, so that `mult` is called with different calling contexts (Figure 5.1 below). We analyze it with different numbers n of noise symbols for input x. Note that even though the interval abstraction for x is always $[-1, 1]$, the relational information between the arguments of `mult` evolves.

We now compare the summary-based modular analysis to the classic zonotopic iterations when n evolves (see Figure 5.1). We see the much better performance of the modular analysis when the instantiation succeeds: it is linear in the number n of noise symbols, whereas the non-modular one is quadratic, because of the multiplication. This is of course a toy example, where increasing the number of noise symbols brings no additional information. But in the analysis of large programs, which is the context where modularity makes the more sense, we will often encounter this kind of situation, with possibly many noise symbols if we want a fine abstraction of correlations and functional behavior (sensitivity to inputs and parameters). This complexity gain is thus crucial. Note that the results for the modular analysis are still very accurate: we obtain, for $y1$, $[-6.5, 2.5]$, to be compared to $[-3, 0]$ with the non-modular analysis, and for $y2$, $[-4, 4]$, to be compared to $[-2, 4]$.

```
int N = 100, i;
int main(...)
  { double x ∈ [−1,1];
    double y1, y2;
    for (i=0; i<N; i++)
     { y1 = mult(x+1,x);
       y2 = mult(x, 2*x);
       x /= 2;
     }
  }
```

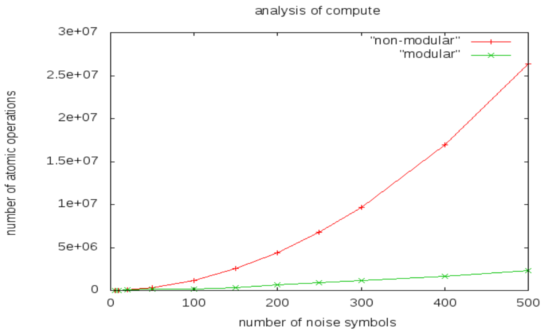


Fig. 3. Number of operations in the analysis, function of the number of symbols

5.2 Application of Summarizing on Benchmarks

We consider the following set of simple benchmarks: *img_filter* is a simple filter that performs edge detection on a small image composed of 20 pixels. The algorithm uses iterative filtering that calls a blur filter followed by 6 calls to a Sobel filter. The application thus filters 40 different descriptions of the initial pixels.

sincos computes an approximation of the sin and the cos functions with different order 5 polynomials, depending on the range of the inputs. The application makes 64 different calls to the function computing the results. At the end of each call, we formally verify that $\sin^2 + \cos^2 - 1$ remains in small ranges.

order2_filter is a linear filter of order two: $S = a * E + b * E0 + c * E1 + d * S0 + e * S1$. The formal parameters of the function are a close to 0.7, b close to -1.3, c close to 1.1, d close to 1.4 and e close to -0.7; inputs E are independent, within $[-1, 1]$. Close to means that the value of these coefficients is only known to be in a range of width 2% of their value. The filter is called 8 times.

The results are shown in Table 1. Even on a rather small number of function calls, we have a significant time gain (at least 5 times as fast as the non-modular analysis) without losing too much precision (worst case being around 2).

	example	*img_filter*	*sincos*	*filter*
Characteristics	#lines of C/#vars	194/135	208/135	96/19
Non-modular analysis	time (s)	11.8	3.84	49
	average interval	$[0.056, 0.067]$	$[-0.026, 0.026]$	$[-1.24, 2.99]$
Modular analysis	time(s)	1.7	0.79	10
	instantiations	18/20	63/65	6/8
	average interval	$[0.056, 0.067]$	$[-0.058, 0.058]$	$[-1.58, 3.33]$
Comparison	time gain	6.9	4.9	4.9
	precision loss	1	2.23	1.27

Table 1. Comparison of modular and non-modular analyses

Our aim is of course to apply this modular analysis to real industrial control software for numerical validation. The applications we target are large reactive systems. Most of the source code for these applications is generated automatically from high-level synchronous data-flow specifications written in SCADE or SIMULINK. These languages allow programming the control software in a highly hierarchical way, with many calls to different levels of blocks. The structure of the C source generated in such a way is one main function that calls many numerical blocks, generally iterating on a large number of cycles.

We report here on a partially manual, partially automated simulation of our method to a real industrial test case, part of a control command software used in the aeronautics industry. This program is about 37500 lines of C, consisting of an infinite loop. The core of the loop first updates the inputs with the sensors' data and then calls a function composed of eight different blocks. We unroll this loop 6 times here, after which the ranges are stable. The program has about 20 input variables, more than 500 sensor variables, more than 10000 constant and local variables, and about 30 output variables. The automated part was done on an interactive version of FLUCTUAT [DGP+09], but we manually simulated the instantiation and call mechanisms. We did not use our standalone prototype here since the code contained features (in particular arrays), that are not treated in the prototype we specifically developed for this article.

A summary is built for the whole function in the loop, and it is reused or updated if necessary by the next iterations. The input summary has 70 variables, the output summary has 90. The analysis takes about 10 minutes for each cycle / function call on a standard Linux desktop. The summary applications is immediate (less than 1 second), and only one summary creation is needed here. For the 6 iterations, the analysis takes 60 minutes without summaries, and 10 minutes with summaries. The time gain may be less impressive than on the smaller examples, but it depends on the structure of the program, and this one is not especially modular. Also, if more loop iterations were needed (thus more function calls), the gain would of course have been higher. The final results are similar with and without summaries; only some partial results are less precise with summaries, but the loss of accuracy is always within 20%.

6 Conclusion and Future Work

We showed in this paper that zonotopic abstractions are particularly well suited as a basis for modular static analysis, by the fact that they form a natural parameterization of input-output relationships between program variables. The algorithm we presented and tested is both simple and efficient. Future work includes the proper testing and improvement of the dynamic partitioning extension to our algorithm (Section 4) and the combination of this numerical modular abstract interpretation together with modular alias analyses. One possibility is to use recent work on shape analysis [RC11], that eases such combinations.

Acknowledgement

This work was funded by CEA Carnot program and ANR projects ASOPT and DEFIS (grants ANR 2008 SEGI 023 02 and ANR 2011 INS 008 05).

References

[Bee06] R. Beezer. *A First Course in Linear Algebra.* available at `http://linear.ups.edu/online.html`, 2006.

[Bou92] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *J. Funct. Program.*, 2(4):407–423, 1992.

[CC77] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CC02] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *CC'02*, LNCS 2304, pages 159–178, Grenoble, France, April 6-14 2002.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'78*, pages 84–96. ACM, 1978.

[CS93] J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. *SIBGRAPI'93*, 1993.

[DGP$^+$09] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS'09, LNCS 5825*, pages 53–69, 2009.

[GGP09] K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain taylor1+. In *CAV'09, Grenoble, France*, 2009.

[GGP10] K. Ghorbal, E. Goubault, and S. Putot. A logical product approach to zonotope intersection. In *CAV'10*, LNCS 6174, pages 212–226, 2010.

[GP06] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *SAS'06, LNCS 4134*, pages 18–34, 2006.

[GP09] E. Goubault and S. Putot. A zonotopic framework for functional abstractions. *CoRR*, abs/0910.1763, 2009.

[GP11] E. Goubault and S. Putot. Static analysis of finite precision computations. In *VMCAI'11, LNCS 6530*, pages 232–247, 2011.

[GT07] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP'07*, volume 4421 of *LNCS*, pages 253–267, 2007.

[JGR05] B. Jeannet, D. Gopan, and T. Reps. A relational abstraction for functions. In *Int. Workshop on Numerical and Symbolic Abstract Domains*, 2005.

[Log07] F. Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In *VMCAI'07*, 2007.

[Min01] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Symposium on Programs as Data Objects, LNCS 2053*, 2001.

[MOS04] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL'04*, pages 330–341. ACM, 2004.

[QR04] S. Qadeer and S. K. Rajamani. Summarizing procedures in concurrent programs. In *POPL'04*, pages 245–255. ACM Press, 2004.

[RC11] X. Rival and Bor-Yuh E. Chang. Calling context abstraction with shapes. In *POPL*, pages 173–186. ACM Press, 2011.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*, pages 49–61. ACM, 1995.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data-flow analysis. In *Program Flow Analysis: Theory and Applications*, 1981.

[SRH96] S. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TCS*, 167:131–170, 1996.

[SSM05] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI'05, LNCS 3385*, pages 25–41, 2005.

[YYC08] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *POPL '08*, pages 221–234. ACM, 2008.